An Architecture for *Circus* Tools

Leo Freitas[1], **Jim Woodcock**[1], **Ana Cavalcanti**[1]

[1]Department of Computer Science, University of York YO10 5DD, York, UK

{leo,jim,Ana.Cavalcanti}@cs.york.ac.uk

Abstract. *Circus is a concurrent language tailored for refinement that combines Z, CSP, and the refinement calculus using Hoare and He's Unifying Theories of Programming (UTP). In this paper we present our architecture that extends an ongoing effort of Community Z Tools (CZT) that implements tools for Standard Z. This includes: a specification processing front-end that allows parsing, pretty-printing, and typechecking of Circus; a theorem proving module; a compiler useful for animation and prototyping; and a refinement model checker.*

1. Introduction

Community Z Tools (CZT) is an initiative started in 2002 that is geared towards providing extensible tool support for Standard Z [ISO/IEC 13568 2002] in Java. Since then, some Z extensions have been implemented. In this paper, we detail how we have built on the top of CZT's architecture a foundation for tools to a Z extension: *Circus*. We also detail a set of tools built from this foundation, as well as other tools that benefited from this architecture. The point is to show a successful case study on how one could use CZT's foundation to effectively produce Standard Z compliant tools.

Circus [Woodcock and Cavalcanti 2001] is a concurrent language tailored for refinement that combines Standard Z with CSP [Roscoe 1997] and the refinement calculus [Morgan 1994], where its semantical background is based on the Unifying Theories of Programming [Hoare and Jifeng 1998]. That is, syntactically the language resembles both Z and CSP, as well as the constructs from the refinement calculus. Semantically, it follows Hoare and He's denotational theory of programming that enables theoretical combination and extension of various paradigms. Despite the architecture explained in this paper being instantiated for *Circus*, many of the techniques explained later on can be reused within CZT for different extensions, hence most of the ideas presented here are reusable in different contexts, as we motivate throughout the paper. If one wants to reuse CZT for extending a new Z-based language, this paper serves as a guideline. That is, the various modules presented here could be adapted to other circumstances and languages.

In this paper, we present four modules that are used as building blocks for various *Circus* tools, some of which we mention along with the module descriptions. In the next section we present a specification processing module that represents *Circus* as an abstract data type, and provides basic functionality like parsing and typechecking. It is the foundation for all *Circus* tools. After that, in Section 3 we define a general theorem proving architecture used by the *Circus* tools to discharge verification conditions (VCs). In this process, the user is given greater flexibility in how the architecture can be fine-tuned, which is important to get higher-levels of automation. This is achieved either via configurable interfaces or integration with external tools. We also mention how we integrate the architecture with the **Z/Eves** theorem prover [Meisels et al. 1997]. Next, in Sections 4 and 5 we present more specialised modules related to compilation and refinement search. The former builds an automaton representing *Circus* specifications from

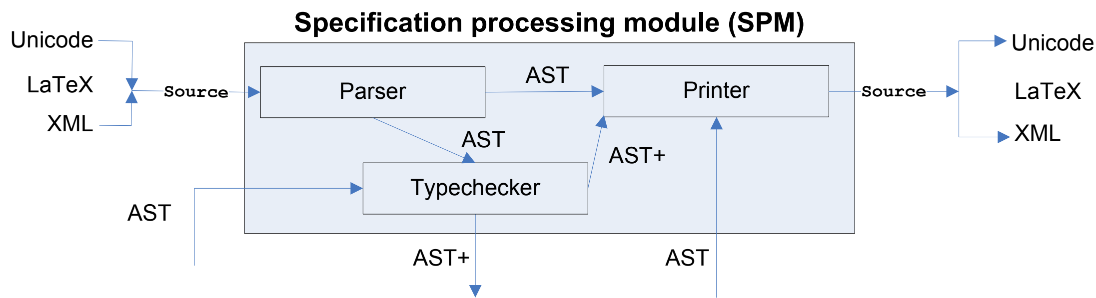

Figure 1. *Circus* **SPM design**

the *Circus* AST produced by the specification processing module, whereas the later performs an exhaustive search over this automaton in order to establish refinement or some properties of interest.

Related work. CZT supports other extensions with parsing, pretty-printing and typechecking, such as Object-Z [Smith 2000]—an object oriented flavour for Z— and TCOZ [Qin et al. 2003]—another Z extensions that includes time and concurrent aspects to Object-Z. Details on how to use CZT for new extensions are available in [Miller et al. 2005]. In [Xavier et al. 2006] a *Circus* typechecker was implemented but not within the CZT framework. Also, in [Freitas and Cavalcanti 2006] a translator tool from *Circus* to Java was built.

2. Specification processing module (SPM)

This module comprises a data type representing *Circus* programs together with a parser, pretty-printer, and typechecker, as shown in Fig. 1. Programs can be given in different markup formats. A *Circus* font and LaTeX style macros extending the Standard Z [ISO/IEC 13568 2002] ones from CZT are also available.

Circus **AST.** Following the design decisions in *Community Z Tools* (CZT) [Malik and Utting 2005], we defined an XML markup format for *Circus* that extends the existing one for Z[1]. The idea of using XML for Z has also been explored in [Meisels et al. 1997]. XML is used as an interchange format to exchange parsed specifications between sessions and tools. An XML format is a good choice, as it is trivial to write parsers for XML, in contrast with the complexity of writing parsing tools for LaTeX. This XML format is the basis of an *Annotated Syntax Tree* (AST) data type represented as Java classes and interfaces.

To manipulate ASTs, we provide a customised interface for each *Circus* construct. These interfaces together with their implementations are automatically generated from the *Circus* XML using XML manipulation tools. For instance, apart from the usual implementation using the *Factory* design pattern, it also produces a more efficient *Fly-weight* version [Gamma et al. 1995] useful in some scenarios, but this is still an experimental feature. Although these *Circus* AST classes shall seldom change, in the case where the language is to be extended with new constructs, or amendments are requested upon demand, modifications are done centrally in a single XSD file. This strategy dramatically reduces the time required to develop extensions, ensures a common style of interface,

[1]This Z XML is called ZML (see `http://czt.sourceforge.net/zml`).

and improves maintainability and documentation. Modularity is also guaranteed by XSD importing: the *Circus* XSD imports the Z XSD. Various traversal strategies for the AST are available: they use a variation of the visitor design pattern [Gamma et al. 1995]. This Visitor has been described in detail in [Malik and Utting 2005]. It is a variant of the *acyclic* [Martin 1997] and the *default* [Nordberg III 1997] visitor design patterns. Advantages over the standard visitors are that it allows the AST interfaces and classes to be extended without affecting existing visitors, and that it allows a visitor to take advantage of the AST inheritance relationships, hence achieving greater levels of code reuse. This basic architectural elements provides *Circus* with: (i) choice of coding style for generic low-level algorithms, and other term-specific high-level algorithms via the different visiting mechanisms; (ii) automatic code generation of AST interfaces and implementations; (iii) reuse through the CZT visitor allowing traversal algorithms to be recycled in flexible ways; and (iv) extensible ASTs. In total, there are 847 Java classes (and about 33*Kloc*—thousand lines of code) representing the complete *Circus* AST. The *Circus* XSD file is about 4.8*Kloc*, where 2.2*Kloc* are imported from the Z XSD. From this, we generate 33*Kloc* (*i.e.,* 85% of the AST classes are automatically generated).

This exemplifies how CZT allows great generality and extensibility without the burden of a high-level of maintainability. The XML file generates one abstract interface with two or three concrete implementations for varied purposes. For instance, one with plain AST classes, and another with the Flyweight pattern for smaller memory footprint. For a new extension, one only needs to create a new XSD file extending either the Z or *Circus* ones. In fact, colleagues in Brazil have just used this strategy to extend *Circus* with an inference rules engine implementing the *Circus* refinement calculus.

***Circus* parser.** A *Circus* parser is used to instantiate the AST representing *Circus* specifications. This is the starting point for all *Circus* tools. The parser is built using a LALR grammar and the Java CUP parser generator [Appel 97]. Similarly, Java Flex is used to generate scanners[2]. This follows the CZT design for Standard Z. This separation of concerns leads to great levels of modularity, reusability and extensibility, which contributes towards good software engineering practice taken throughout the whole development.

Code from parser generators like CUP is not easily reusable. So, we use XML again for defining parser and scanner templates that enable a more efficient way of inheriting object-oriented code from the Z parser. The XML templates are transformed into a series of Java (`.java`), CUP (`.cup`), and Flex (`.jflex`) files that are in turn compiled into the final parser/scanner Java code. These different XML files representing the parsers and the scanners are processed using XSLT, a language for transforming XML documents. This maximises the commonality between the parsers/scanners, and minimises versioning and maintenance problems. The whole parser code is around 30*Kloc*, and it is generated from around 16.5*Kloc* of XML template files (*i.e.,* 43% of generated code). We also provide support for pretty-printing from an AST to any of the available markup formats. This is particularly useful when using different tools in the pipeline that do not understand XML, but do understand LaTeX or Unicode. On the other hand, the XML printer can be used to enable interoperability among different programming languages implementing tools for Z and/or *Circus*.

[2] See `www2.cs.tum.edu/projects/cup/` and `http://jflex.de/`.

A major difference in the *Circus* parser from the original CZT design is in error recovery. The user can control the parsing depth for fatal errors, as well as follow context-sensitive error messages. The latter is very useful when users are not familiar with the idiosyncratic typesetting aspects of standard Z (and *Circus*). The technique is quite simple, yet very effective: introduce spurious production rules throughout the LALR grammar wherever common mistakes are likely to happen. So, instead of a general "`Syntax Error`" message, the user gets a series of context-based errors depending on which spurious (erroneous) production the error passed through. As those spurious productions introduce no parsing conflict, they cannot recognise correct specifications, hence represent a safe and sound solution for handling parsing errors. And this is a general strategy that can be reused for different parsing tasks using LALR grammars parser generators. We also support a warning system hinting to the user potential mistaken and/or less-common specification choices that are, however, syntactically correct. This follows the trend of modern compilers for complex programming languages. For instance, the parser/typechecker handling generics in Java 1.6 warn the user in the case of potentially dangerous run-time type cast that might be entirely correct/appropriate.

Multiple markups. The parser supports multiple markups, as different markup languages suit different communities. For example, LaTeX is preferred by researchers, while Unicode WYSIWYG editing might be more attractive for students or industrial users. At present, Unicode, LaTeX, and XML formats are supported. In order to avoid having to provide a parser for each markup language, all specifications are first translated into Unicode and subsequently parsed by a Unicode parser[3]; the AST is markup independent. This is a necessary precondition for allowing different sections of a specification to be (possibly) written in different markups. If a parser for a new markup is required, only a translator to Unicode is needed.

In order to support LaTeX markup, it is convenient to provide a LaTeX toolkit for a given extension that defines new operators and keywords for that language; this forms a LaTeX markup section [ISO/IEC 13568 2002, Ap. B]. In addition to defining new operators, these LaTeX markup documents contain LaTeX markup directives used to specify how certain LaTeX commands are to be converted into Unicode. For *Circus*, this section of keywords, operators, and other directives, are defined in the `circus_toolkit.tex` file. And the users can layer their *Circus* (or Z) specifications in a similar (modular) fashion through section inheritance, which can be multiple but not circular, as defined in the Z Standard [ISO/IEC 13568 2002, Sec. 13.2.2]. The LaTeX to Unicode translator parses these definitions and converts each LaTeX command into the corresponding Unicode sequence. Nevertheless, for new LaTeX environments (*i.e.,* `\begin{xxx}` and `\end{xxx}`), we cannot use LaTeX markup directives. Instead, the LaTeX to Unicode converter needs to be adapted directly. This is possible if we add new rules to the converter XML template. The LaTeX directives/commands defined in `circus_toolkit.tex` are typeset using LaTeX macros defined in the `circus.sty` style file, which enables correct LaTeX typesetting for *Circus*. The definitions of these LaTeX macros in `circus.sty` rely on selecting the appropriate Unicode character from the font where they come from. That means we also need to provide a *Circus* font. We extend CZT's true-type, metafont and Adobe Type1 fonts for *Circus*. An additional benefit of following CZT's approach

[3]See [Malik and Utting 2005] for a more detailed description of the parser architecture.

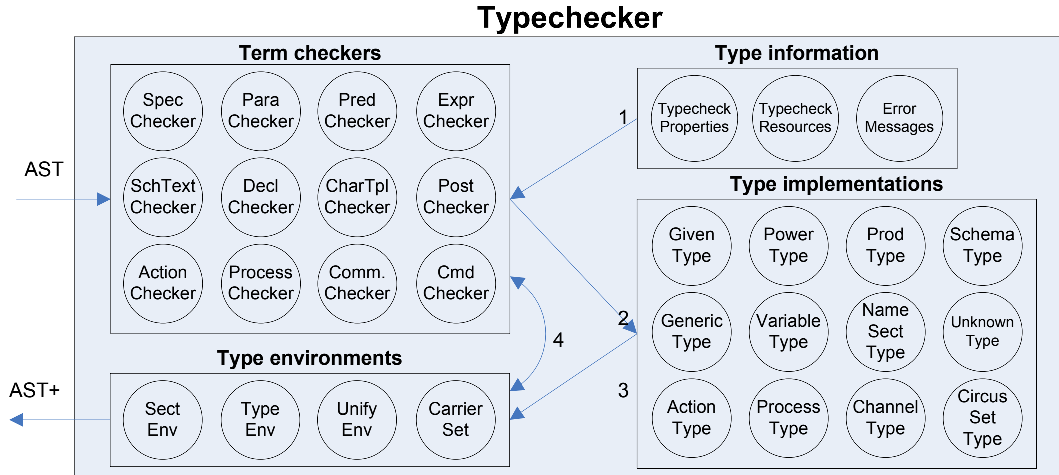

Figure 2. *Circus* **typechecker design**

using intermediate Unicode is that we reduce the number of converters needed between markups to be implemented from $M * (M - 1)$ to $2 * (M - 1)$, where M is the number of markup languages supported.

Typechecker. We extend CZT's Z typechecker with *Circus* typing rules [Xavier et al. 2006]. As the parser provides an AST as a Java object, each extension has its own typechecker. So, using XML templates is unnecessary because unlike the parsers, Java interfaces and inheritance can be used to extend the Z typechecker. When a *Circus* specification AST is typechecked, all the typechecking rules are applied, and if the specification is type-correct, it annotates the original AST with type information as defined in the ISO standard [ISO/IEC 13568 2002, Sec. 10]. If the specification contains type errors, a list of error messages describing them (including their line and column positions) is available.

Most of the typechecker is written using the visitor pattern. While it is tempting to write it as one large visitor, this would create maintenance problems as this visitor would be quite large and monolithic. So, we (re)use the rather elegant and more sophisticated CZT design extended for *Circus*, as shown in Fig. 2. This architecture using individual checkers per different AST syntactical categories breaks up the overall task of typechecking into several (currently nine) smaller `Checker` visitors — each subclass of `Checker` typechecks a different kind of syntactic construct, such as paragraphs, predicates, expressions, *etc*. The `Checker` class itself defines some shared resources, such as typing environments and the type unification facilities, as well as common "helper" methods used throughout the implementation, such as error reporting. In addition, each checker subclass object has a reference back to the top-level `TypeChecker` object, which has links to all the checkers — this allows one checker to call (the right version of the) other `Checker` via the `TypeChecker` object. In this way, the typechecker design is much like promotion in Z specifications [Woodcock and Davies 1996, Ch. 13].

For example, for typechecking a schema text $[\, Decl \mid Pred \,]$ of an `AxPara` term representing a Z schema, the `ParaChecker` class, which typechecks Z paragraphs, needs to typecheck both the declaration and predicate parts of the schema text. Although visiting through the given AST is the general solution, the typechecking is delegated to

the `DeclChecker` and `PredChecker` classes, respectively. The `DeclChecker` in turn uses the `ExprChecker` to ensure that expressions defining the declaring variables type are well-formed. Because each of these visitors share the same `TypeChecker` reference, and hence the same references to the type environments, declarations added to the type environment by the `DeclChecker` will be accessible by the other checkers. Successful typechecking returns an AST^+ annotated with type information for each of its relevant terms with a general `SectTypeEnv` class containing type annotations [ISO/IEC 13568 2002, Sec. 10]. Other constructs are decomposed accordingly. There are a few additional classes that are used by the typechecker: a type environment containing all known types and variables; a `UnificationEnv` class that performs unification of two types for type inference and for checking type consistency; and so on. This splitting of the overall typechecking task into several visiting parts increases modularity and maintainability, and provides better encapsulation for the different checkers. This aids debugging, and allows development of the checkers to be somewhat independent (*e.g.,* assigned to different teams).

An interesting feature of this design is that each `Checker` class can be extended and plugged into the main `TypeChecker` class independently. For instance, the `DeclChecker` for ***Circus*** extends the Z version with qualified formal parameter declarations that reuse the Z `PredChecker` but needs to extend the `ExprChecker` for the extended expressions present in the ***Circus*** grammar (*i.e.,* channel set expressions used in parallel composition). Another advantage is that each `Checker` is typechecking similar kinds of terms (*e.g.*, expressions), so can have a uniform visiting protocol, which increases regularity, and helps to reduce coding errors. For example, all the visitor methods of the `ExprChecker` class typechecks expressions, and all its visitor methods return a `Type` term with resolved reference parameters in which type unification and generic actuals inference have already been performed.

3. Theorem proving module (TPM)

The module is unrelated to (and independent from) the CZT design decisions. And when combined with the parsing and typechecking capabilities of the SPM (see Fig. 1), it can be used as an independent tool. It is an extended version of the prototype described in [Freitas 2005, Ch. 5]. The TPM discharges *Verification Conditions* (VCs) generated by other ***Circus*** tools or input directly by the user, such as the compiler or refinement model checker described below. For instance, VCs can be predicates to be discharged as a result of some term manipulation, such as the application of refinement laws. Firstly, a type correct AST^+ representing the VC is passed to the symbolic-set utility tools, which perform syntactic transformations over the AST^+ in order to try to evaluate expressions, solve predicates, or manipulate schemas. For instance, they can exploit some set theoretic laws that are syntactically recognisable, such as the zero laws for set union (*i.e.,* $A \cup \emptyset = A = \emptyset \cup A$). Within the symbolic-set utilities (see Fig. 3), one can attach their own syntactic evaluation facility. In fact, those evaluators are layered with different implementations in an "*onion-like*" shape, where outer layers syntactically transform the term, and then pass it along to the inner layers (if any). For instance, suppose we are using a 3-layer shape with relational, prepositional, and predicative operator transformers. A term would first pass through the relational layer, where all relational operator application or set containment expressions would be modified. Quantifiers (or quantified set operations like $\bigcup$) are left

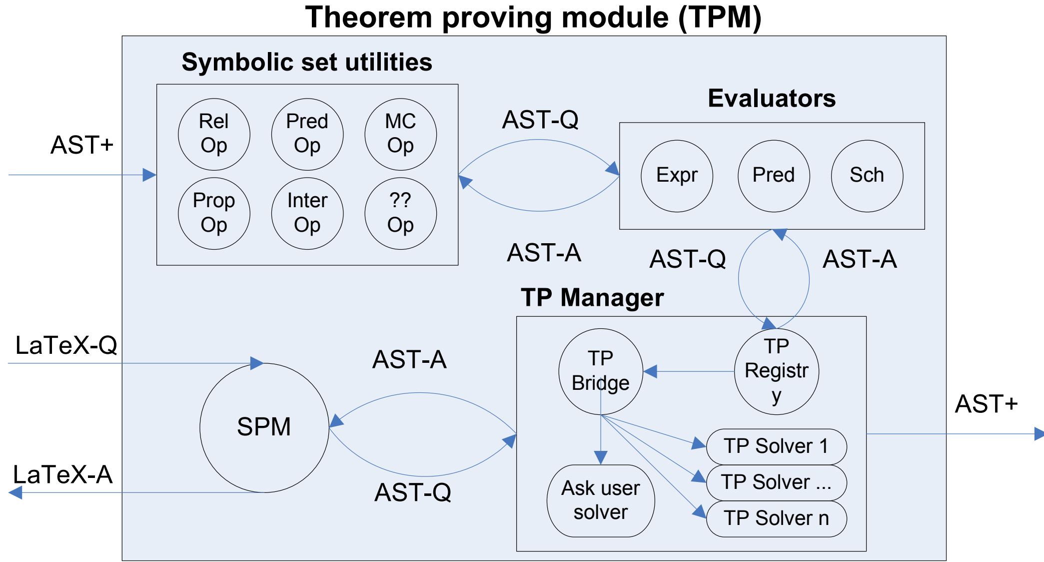

Figure 3. *Circus* **TPM design**

```
public interface RelationalSetOp {
    public Pred isMember(Expr e, Expr S) throws ...;
    public Pred subset(Expr S, Expr T) throws ...; }
public interface PropositionalSetOp {
    public Expr union(Expr a, Expr b) throws ...;
    public Expr intersection(Expr a, Expr b) throws ...; }
```

Figure 4. Some TPM set utility operations interface by category

unchanged until they reach the predicative inner-layer. In Fig. 4, we present some of the interfaces of these transformers. A similar, yet simpler, set of interfaces is also available for the case where only finite (enumerable) sets are used. In this case, we have not only more automation, but also other complex operations, such as power set construction.

For each (of the five currently available) syntactic-set utility interfaces, the user could mix any of the five layers of syntactic transformer implementations. That is, we have five different implementations of the relational operations layer, and so on for the others in Fig. 3. The user could also define new interfaces if needed. Each layer may (or may not) transform the term, depending on the term's structure. For instance, for the `union` operation from the `PropositionalSetOp` interface above, the available implementation ("*onion-like*") layers are ordered as: (i) *plain sets* that directly apply the given operation (*i.e.*, `union(s1, s2)` $=$ $(s1 \cup s2)$); (ii) *set extension* that exploits the set structure (*i.e.*, `union({e1, e2}, {e3})` $= \{e1, e2, e3\}$); (iii) *Z Toolkit* laws (*i.e.*, `union(s1, {})` $= s1$); (iv) *typechecked VCs*, when expression transformation is needed; and (iv) *definition expansion* (*i.e.*, `union(s1, s2)` $= \{e : \sigma T \mid e \in s1 \lor e \in s2\}$), where σT represents the unified carrier set (or unified maximal type) from the `union` elements. Default settings are in place so that nothing is required from the user at first. Nevertheless, fine-tuning (or extending) the available facilities proves helpful in increasing levels of automation when dealing with specific kinds of expressions and predicates. That is, as the disposition of these layers affects the efficiency and au-

tomation levels, the user is given the choice on how to accommodate them, with default values in case this is not of concern. For instance, for model checking, we added two new layers (with the five above implementations each) for model checking operations (*e.g.,* `MCOp`) as well as interactive operations like nondeterministically choosing an element from a set (*e.g.,* `InterOp`). As we can introduce additional interfaces (*i.e.,* skin layers for our "*onion-like*" structure of transformers) with different implementations (*i.e.,* different skin "colours" for each layer), greater generality is possible. It also opens a wide range of opportunities for fine-tuning the TPM for VCs from different application domains. In general, as we can have as many "*onion-skin*" layers with as many possible implementations as we like/need, the general formula for possible combinations is given as $\sum_{i=0}^{n-2}(\frac{n!}{i!} \times m^{n-i})$, where n is the number of possible "*onion-skin*" layers or set utility interfaces, and m is the number of available implementations for each n_i layer. Currently, with five layers ($n = 5$) and five implementations each ($m = 5$), there are $458,025$ possible combinations for fine-tuning the *TP Manager* for VC evaluation. Yet, many of these permutations are commutative, so the actual number is considerably reduced in practice. An interesting research opportunity is to find a taxonomy of permutations tailored for different application domains that would lead to higher-levels of automation. So far, we have done this only for a small variety of case studies.

The AST$^+$ is passed to the symbolic evaluators, which perform semantic transformations over expressions, predicates or schema texts, in order to try to automatically simplify them. For instance, a trivial evaluator of expressions and predicates is available, wherever constituent parts are numbers or enumerated sets. This trivial evaluator can cope with arithmetic expressions, and some relational predicates involving numbers, symbolic variables and constants. It forms the basis of a simple propositional calculus SAT solver. All tools using TPM are provided with benchmarking facilities in terms of CPU execution time, memory requirements, number of (different) VCs (by category), and so on.

These VCs AST$^+$ in the TPM can be interpreted as questions (AST-Q) that expect some simplified answer (AST-A). Upon termination, if the predicate or expression still require manipulation (*i.e.,* a predicate is not *true*, or an expression is not a number), it is passed to the *TP Manager* in order to try one of the stacked solvers. Each stacked solver represents a tool trying to discharge one of the transformed VCs. The *TP Registry* is a pool of available tools to chain the request for discharging the VCs, whereas the *TP Bridge* is a minimal Java interface all external tools must satisfy in order to be used by the TPM (see Fig. 5 below). Ultimately, if after both syntactic manipulation of terms from the symbolic-set utilities, and semantic manipulation of terms from the symbolic evaluators, terms are not fully simplified, a final low-level evaluator is used. This evaluator is always enabled and translates the current AST-Q into LaTeX (or other chosen markup format), presenting the result to an assigned output stream, usually the standard output (*i.e.,* LaTeX-Q). That means the user is challenged with a query about the truth of a predicate, or the actual value of a expression, for which a corresponding answer (in the selected markup format) is expected (*i.e.,* LaTeX-A). Similarly as before for the symbolic-set utilities, we have symbolic-evaluator interfaces for semantic transformation together with some available implementations. These entities can again be layered up in an "*onion-like*" structure used for fine-tuning semantic transformation of VCs passing them through the TPM.

```
public interface SymbolicEvaluator {
    public boolean setProperty(String name, Object value);
    public Object getProperty(String name) throws ...; }
public interface ExprEvaluator extends SymbolicEvaluator {
    public Expr evaluate(Expr expr) throws ...; }
public interface PredEvaluator extends SymbolicEvaluator {
    public Pred evaluate(Pred pred) throws ...; }
public interface SchExprEvaluator extends ExprEvaluator {
    public ZSchText evaluate(ZSchText term) throws ...; }
```

Figure 5. *TP Bridge* minimal Java interface

Theorem proving bridge. The minimal Java interface to implement in order to integrate with the stacked solvers is given in Fig. 5. The `evaluate` methods are called by the *TP Manager* whenever some internal evaluator could not cope with the term being passed as a parameter. In fact, because of this stacked solvers interface, the user could implement or integrate other evaluation tools. For instance, the user could have extra symbolic-evaluators implementations as Java libraries. The `get/setProperty` methods can be used by the tools to get/pass useful information. To cope with the schema calculus, a particular feature of Z (and *Circus*), we also have an evaluator for schema texts (*i.e.,* `[ Decl | Pred ]`).

Integration with tools. From CZT, we borrow three internal set evaluation (Java) tools: a schema unfolder that is useful to simplify VCs containing schema texts into a predicate or expression; the ZLive Z animator, which allows symbolic evaluation of certain (finite-type) VCs; and a rewriting engine for Z that resembles a simple natural deduction tactic language. They are all available from CZT. The schema unfolder is useful for normalising the schema calculus to a minimal subset of operations, which makes precondition calculation and other schema calculus operations easier to perform. The rewriting engine enables one to specify, in any of the available markups, natural deduction rules, which allows a simplified term-transformer tactic language in the spirit of Angel [Martin et al. 1996]. Finally, ZLive is an animator for Z specifications that is a revised extension of Jaza [Utting 2005], which follows constraint solving principles. To integrate external tools, one needs to perform three setting up tasks: (i) support at least one of the available parsing markup formats; (ii) implement the minimal Java interface bridge described above; and (iii) recognise set-theoretical expression and predicate terms, where manipulation of Z schemas is a bonus but not a requirement, since schemas can be previously unfolded into predicate. So far, we have integrated one external tool: the **Z/Eves** theorem prover [Meisels et al. 1997]. The**Z/Eves** back-engine runs as a network service through a socket connection that listens to clients "speaking" their proprietary XML format [Saaltink et al. 2005]. Thus, to connect with this engine, we need to set an IP address and a port for the host machine where the engine is running.

4. *Circus* compiler

The *Circus* compiler implements the operational semantics specified in [Freitas 2005, Ch. 3], and is currently being updated to reflect a newer version of the semantics [Woodcock et al. 2007]; its architecture is detailed in Fig. 6. The operational se-

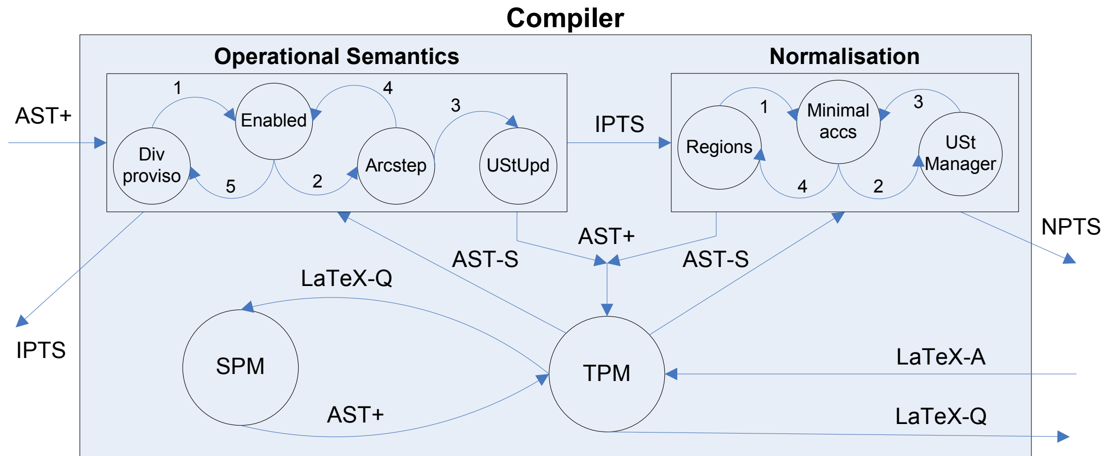

Figure 6. *Circus* **compiler design**

mantics compiles typechecked AST$^+$ trees into the specialised automata theory devised for *Circus* and formalised in [Freitas 2005, Ap. A.3]. Compilation takes place on-the-fly as needed, and it produces either an *IPTS* (*Implementation Predicate Transition System*), or a *NPTS* (*Normalised Predicate Transition System*) object representing the semantics of *Circus*. The normalised *IPTS* (*NPTS*) is a deterministic automaton used during model checking as the specification side of a refinement check. Normalisation is important during model checking as it ensures that possible counterexamples obtained are the smallest possible, that their paths representing failures are unique, and that debugging information after the refinement search can be purposefully cast into human-readable format with the least effort possible. For mode details on this, see [Freitas 2005, Ch. 3–4]. If model checking is not intended, and only process exploration is used, no normalisation is required. Like the TPM, the compiler exists as an individual tool that can be used to explore the behaviour of *Circus* process, much like ProBE [Probe 2000] does for CSP [Roscoe 1997].

During compilation VCs might be generated in order to identify possible new paths in the underlying specification, for example, when guards from input prefixing need to be evaluated. These are compilation VCs (or cpVCs) and are usually application dependent, as they come directly from the *Circus* semantics. That is, the more complicated the data types within a *Circus* specification, the more complicated the cpVCs generated by the compiler. Also, these are the VCs closest to the kind of theorems one would be proving about the specification anyway, hence some form of automation libraries might already be available, or the user would tend to be familiar with the shape of the VC.

The operational semantics rules are implemented by two different visitors, namely `Enabled` and `ArcStep`, together with two additional auxiliary visitors, namely `DivProviso` and `UStUpd`. The `Enabled` visitor returns sets of type correct communication expressions for each *Circus* AST$^+$ term: it calculates the set of immediately available events for every *Circus* construct that will form the arcs of the *IPTS* automaton. The `ArcStep` visitor builds up the *IPTS* nodes reachable from previously calculated `Enabled` arcs. It returns a list of `INodes` (*i.e., IPTS* implementation nodes) for each *Circus* construct, provided one of the immediately available communicating events returned by `Enabled` is chosen. The `UStUpd` visitor performs updates on the state of the current process being compiled, depending on the *Circus* construct used. Finally,

the `DivProviso` visitor is an experimental feature for "*early divergence*" (or rather explicit divergence) detection and propagation caused by some language constructs. In *Circus*, early divergence can be detected (and propagated) from its basic constructs, such as `Chaos`, `SchExpr` (outside their preconditions), and so on. These provision conditions are essentially quite simple conjoined (boolean) predicates that, when evaluated to *false*, would characterise divergent behaviour. On the other hand, implicit divergence usually caused by hiding too many events or via unguarded recursion, cannot benefit from this strategy. Instead, implicit divergence is detected just like in the FDR refinement model checker [Goldsmith 2000] for CSP: silent loops in the *IPTS* representing the specification. Silent loops are represented by an arc containing the empty set (*i.e.,* $\mathtt{Enabled} = \{ \emptyset \}$), whereas deadlock is represented by no enabled arcs at all (*i.e.,* $\mathtt{Enabled} = \emptyset$). As divergence detection is a bottleneck in FDR's performance due to its implementation need to use depth-first search, our early (or explicit) divergence detection scheme might pay off in performance gains. Furthermore, it turns out that these conjoined predicates built by the `DivProviso` visitor might also be useful in calculating some data independent abstractions of infinitely recursive (yet finite-state) processes.

The normalisation sub module performs semantically-equivalent transformation of compiled *IPTS*, so that they become deterministic (*NPTS*). As *IPTS* arcs represent (possibly infinite) sets of communicating events, rather than a single event as in FDR automata, normalisation is more than mere textbook subset construction [Hopcroft et al. 2001], and it becomes a non-trivial task. Essentially, we need to build the normalised arcs as the individual regions of (possibly nondeterministically) defined communication expressions. For example, the compiled *IPTS* for action A below contains two arcs with an interleaved enabled (communicable) value ($c.1$) yet different refusal sets for each path.

$$\mathbf{channel}\ c : \{ 0, 1, 2 \}$$
$$A \mathrel{\widehat=} (c?x : (x \in \{ 0, 1 \}) \rightarrow A) \Box (c?y : (y \in \{ 1, 2 \}) \rightarrow A)$$
$$\mathit{enabled}\,(A) = \{ \{ c.0, c.1 \}, \{ c.1, c.2 \} \} \quad \mathit{refusals}\,(A) = \{ \{ c.2 \}, \{ c.0 \} \}$$

Thus, to make it deterministic we need to separate the three possible `regions` of events, namely: those unique to the left (*i.e.,* $(c.0)$) and right (*i.e.,* $(c.2)$) sides, respectively; and those shared (*i.e.,* $(c.1)$) by both sides. The `Regions` sub module in the compiler implements a general transformation strategy for situations like this, which boils down to generalised disjointness of all possible subsets (*i.e.,* power sets) from the results of `Enabled (A)`. This semantic-equivalence preservation also takes care of failures information through the calculation of minimal acceptances (or maximal refusals) from (possibly) different nondeterministic paths. Again, like the `UStUpd`, the `UStManager` visitor is responsible for the specialised rules for joining the state of the collapsed automata nodes during the determinisation procedure.

Although the architecture of Fig. 6 is tailored for *Circus*, we believe it can be reused for other state-based languages that is represented with some sort of automata theory [Hopcroft et al. 2001], and wish to integrate automata generation with theorem proving capabilities. Only normalisation is specifically related to the automata theory for *Circus*. Still, whenever deterministation of a more general automata is needed, this module would still be useful in general. Thus, we see this architecture as a good starting point for many state-based language and/or process algebra implementation.

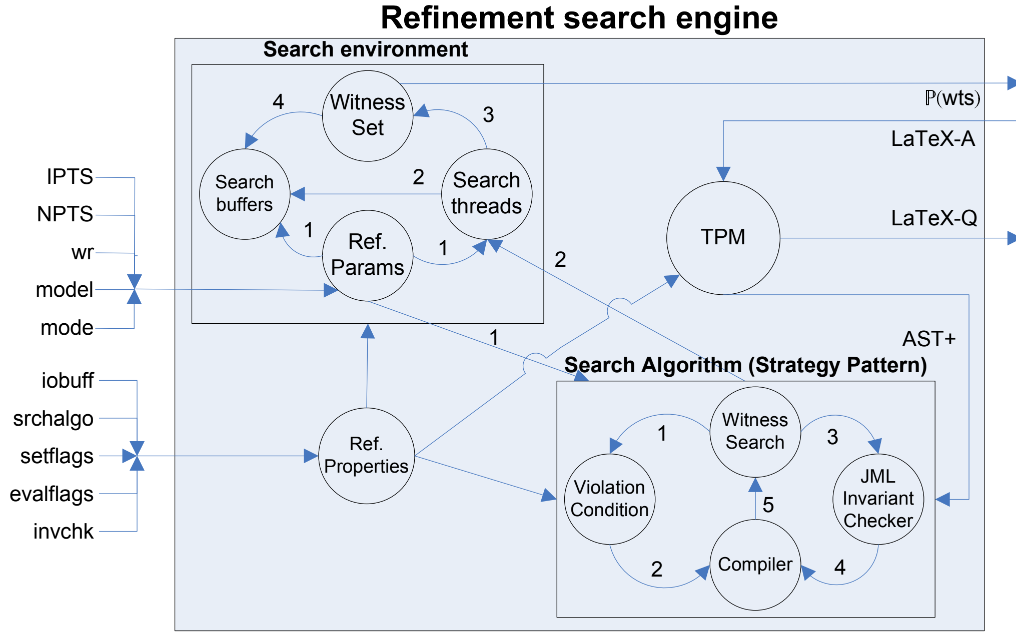

Figure 7. *Circus* **refinement engine design**

5. *Circus* refinement engine

A *Circus* model checker combining refinement model checking with theorem proving was the initial motivation behind the architecture extension of CZT for *Circus*. Taking our own medicine, we decided to use formal verification technique early in the development stage and use *Circus* itself to bootstrap the formal specification of its own verification tool, as explained in [Freitas et al. 2006]. Following this path of development, the refinement engine code was derived using the refinement calculus [Morgan 1994], and few aspects of it have changed since. Those aspects still changing are related to the way VCs are generated and handled by the TPM. The architecture for this module is given in Fig. 7. For every refinement check ($S \sqsubseteq_M I$), the refinement search engine expects two automata: one normalised *NPTS* representing the specification side (S) of the refinement relation; and one *IPTS* automaton for the implementation (I). Other refinement search parameters are also expected, such as: the number (`wr`) of witnesses requested; the `model` ($\sqsubseteq_M$) (or level of detail) to consider; and the search `mode` as model testing (*i.e.,* "error finding"), or model checking (*i.e.,* "correctness checking"). This information is given to the search environment, which runs the actual searching threads and I/O buffers. The user can also configure other global refinement search properties. The `iobuff` defines the appropriate interface with the user through input and output streams, and can be used to pipeline different tools. The `setflags` and `evtflags` are bit masks defining the fine-tuned combination of TPM symbolic-set utilities and evaluators, respectively (see Sec. 3). CZT provides prototype translators from Z to the *Java Modelling Language* (JML) [Leavens et al. 2004]. We used some of these facilities whilst translating to Java the results of the formally derived model checker refinement search algorithm [Freitas 2005, p.121]. The `invchk` flag determines if the invariants calculated from this derived code and translated to JML should be checked. Finally, the `srcalgo` parameter specifies the variation of the search algorithm to use: it acts just like Gamma's *Strategy* design pat-

tern [Gamma et al. 1995]. At the moment we only have a sequential version. Nevertheless, this design leaves open a research opportunity to use more advanced/efficient approaches, such as parallelisation [Orni et al. 1996, Martin and Huddart 2000] or graph-prunning [Dillenburg and Nelson 1993, Fleischer et al. 2000].

During the model checking search process, which is essentially a guided behaviour exploration using the compiler, some VCs related to either normalisation or possible paths to follow are generated. As the refinement engine drives the search process, it feeds back to the compiler the desired "paths" to follow, which in turn need to be derived during the on-the-fly construction of the *IPTS*. These model checking VCs (or mcVCs) are more general, as they are independent from the *Circus* specifications under concern. This makes these VCs more repetitive, which means they are more likely to have a theory useful for all model checking sessions. For instance, we have already built a general theory for the `regions` and minimal acceptances functionality used during normalisation that has shown to be quite useful and reusable. On the other hand, the very fact they are general and application independent leads to the inconvenient aspect that these mcVCs may seem at first unfamiliar to the user. Also, most of these expressions involving `regions` are non-trivial, and require some more powerful theorem proving facilities (and general theories) in order to be automatically discharged. Debugging facilities are yet limited. At the moment they are just user-friendly text messages written to standard output informing the different nature of refinement search failures.

Once again, although this architecture is tailored for *Circus*, it can be generalised to other (state-based) process algebra, provided it is representable using automata theory (*i.e.,* nodes and arcs associated through a transition relation representing a language). That is, given a compiler that feeds the search engine with such automata, and different forms of validation conditions, this architecture could be instantiate for CSP, for example. The difference from FDR would be that it has theorem proving support, which allows more expressive specifications at the cost of potential interactivity via theorem proving.

6. Conclusion

In this paper we present an overview of the architecture of *Circus* based tools that we are developing. It includes some level of detail from all the constituent components. The complete architecture is composed by several modules, from which independent tools can be derived. So far, we have four (prototype) analysis tools: (i) a specification processing front-end that can parse, pretty-print, and typecheck *Circus* programs; (ii) a general (yet rather simple) theorem proving environment that can cope with expressions, predicates, and Z schemas (iii) a *Circus* compiler useful for exploring process behaviour; and (iv) a *Circus* model checker able to perform refinement checks for *Circus*.

These modules are combined in different ways, and four tools have been assembled. The textual interfaces of all these tools have benchmarking facilities in terms of CPU execution time, memory requirements, number of (different) VCs (by category), and so on. **(1)** We have a front-end tool for the SPM with a textual interface, as well as integration with development environments, such as jEdit and Eclipse. It enables the user to validate (*i.e.,* parse or pretty-print, and typecheck) *Circus* programs. When embedded in a development environment, the user can also type the specification while validating it. **(2)** The process explorer enables the user to investigate the behaviour of programs, much

like what the ProBE [Probe 2000] tool does for CSP [Roscoe 1997]. It comprises the specification front-end mentioned above, together with the compiler and the TPM. TPM is needed as the *Circus* operational semantics might generate VCs (cpVCs), depending on the levels of complexity and expressiveness used. **(3)** The model checker tool performs refinement checks of programs, much like what FDR [Goldsmith 2000] does for CSP. It is the most complex of available *Circus* tools using all the modules described in this paper. It combines the two tools above with a refinement engine and a debugger. The refinement engine is a knowledgable explorer during on-the-fly compilation. Essentially, from an initially compiled AST, the engine drives the compiler forward by building the relevant parts of the graph that are important for the refinement claim. In this process, further model checking VCs (mcVCs) might be generated. Such claims can be for either: a property satisfied by a *Circus* specification, like deadlock freedom; or a correctness criteria check while introducing some improved design for the original abstract specification. Whenever witnesses are found, a debugger is used to appropriately present results to the user. **(4)** The VC elimination engine tool composes the specification front-end with the TPM. It is a stand-alone version of the theorem proving facilities available for both the model checker and the process behaviour explorer. It can be useful as plugins to other tools that might handle *Circus*. That is, instead of using our architecture to verify *Circus* programs, we could use other tools for this purpose, where peculiar *Circus* constructs could be handled by this tool. For instance, it could be integrated with the RODIN[4] tools for Event-B [Schneider 2002]. Nevertheless, apart from Z schema text and *Circus* constructs it manipulates, as the VCs it can discharge are general predicates and expressions, the engine could be embedded as a solver for other tools and (possibly) achieve higher-levels automation, provided good application oriented theories are available and appropriate fine-tuning of the TPM configuration are in place.

The next step after further testing is to switch off programming language debugging aids, such as Java assertions and JML static checking, and assemble the various modules adequately for a first beta release of these tools. We have also included some initial support for the jEdit generic development environment with extensions for the available CZT Standard Z plugins for *Circus*. A character map plugin allows one to hit some buttons, where the corresponding character is input into the current opened buffer in the corresponding markup format chosen (*i.e.,* hitting the button with $\mathbb{P}$ power set includes `\power` when in LaTeX markup mode). A tree panel plugin presents ASTs visually so that one can browse through the tree, while the corresponding piece of syntax is highlighted in the corresponding opened buffer. A command-line console plugin enables starting up individual tools, such as the *Circus* process explorer or the model checker, to operate on one of the opened and already processed (*i.e.,* parsed and typechecked) buffers. Finally, an incremental parsing plugin wraps around the *Circus* SPM to enable incremental parsing (*i.e.,* parse-as-you-type) functionality. These plugins are connected with jEdit's error reporting facility, so that errors and warnings for the various tools are displayed accordingly and uniformly. More details can be found at both *Circus* and CZT web-sites[5].

Future work. For the SPM, we need to polish up usage examples. For the TPM, we are currently exploring the ability to tag VCs with manipulation parameters from its point of

[4]See `http://rodin.cs.ncl.ac.uk/Publications/RODIN-Desc.pdf`
[5]See `www.cs.york.ac.uk/circus` and `http://czt.sourceforge.net`.

origin, so that batch (*i.e.,* completely automatic) execution can take place. These parameters would act as “default” (automatic) decisions to be taken. This facility is particularly useful at early stages of development, where the designer is mostly concerned in find early design bugs, rather than proving correctness of desired properties, or some refinement among specifications. We also envisage some sort of low-level VC caching, so that trivial (immutable) results are calculated only once, hence lowering the potential number of user interaction whilst discharging more complicated VCs. Luckily, as VCs tend to follow a pretty repetitive pattern, case studies over particular application domains should shed light on most appropriate tuning options of the TPM for model checking *Circus* programs. We see these facilities being useful for both the process explorer and the model checker. Moreover, we see integration with other external tools as paramount towards efficiency and greater levels of automation while discharging VCs. We have preliminary studies for integration with other interesting tools, such as PVS’s ICS solver [ICS 2005] or ProofPower-Z theorem prover [Arthan 2003], so that we could bridge their term transformation capabilities within our theorem proving architecture from Fig. 3. Finally, an interesting experiment would be to establish a systematic rationale for choosing the appropriate stacked solvers, and/or most efficient TPM flags to increase automation.

Acknowledgements. We received considerable support from many CZT members. In particular, Mark Utting, Petra Malik, and Tim Miller. Ian Toyn provided insight on some obscure Z features. We are also greatful to QinetiQ Malvern for their long term support for the development of *Circus* tools.

References

Appel, A. (97). *Modren Compiler Implementation in Java*. Cambridge University Press.

Arthan, R. (2003). *ProofPower Tutorial*. Lemma-One.

Dillenburg, J. and Nelson, P. C. (1993). Improving the efficiency of depth-first search by cycle elimination. *Information Processing Letters*, 45:5–10.

Fleischer, L., Hendrickson, B., and Pinar, A. (2000). On Identifying Strongly Connected Components in Parallel. In *15th IPDPS*, pages 505–511. Springer-Verlag.

Freitas, A. F. and Cavalcanti, A. L. C. (2006). Automatic Translation from *Circus* to Java. In Misra, J., Nipkow, T., and Sekerinski, E., editors, *FM 2006: Formal Methods*, volume 4085 of *LNCS*, pages 115 – 130. Springer-Verlag.

Freitas, L. (2005). *Model Checking Circus*. PhD thesis, University of York.

Freitas, L. et al. (2006). Taking our own medicine: Applying the refinement calculus to state-rich refinement model checking. In *8th ICFEM*, pages 697–716.

Gamma, E. et al. (1995). *Design Patterns*. Addison Wesley.

Goldsmith, M. (2000). *FDR2 User’s Manual version 2.67*. Formal Systems (Europe) Ltd.

Hoare, C. A. R. and Jifeng, H. (1998). *Unifying Theories of Programming*. International Series in Computer Science. Prentice-Hall.

Hopcroft, J., Motwani, R., and Ullman, J. D. (2001). *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2nd edition.

ICS (2005). *ICS Manual (Version 2.0)*. SRI International.

ISO/IEC 13568 (2002). *Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics*. ISO/IEC.

Leavens, G. T. et al. (2004). *JML Reference Manual*. Iowa State University. v.1.93.

Malik, P. and Utting, M. (2005). CZT: A Framework for Z Tools. In *4th ZB*. Springer.

Martin, A. C. (1997). Acyclic Visitor. In *Pattern Languages of Program Design 3*. Addison Wesley.

Martin, A. P. et al. (1996). A tactic calculus. *FACJ*, 8(E):244–285.

Martin, J. M. R. and Huddart, Y. (2000). Parallel algorithms for deadlock and livelock analysis of concurrent systems. *Communicating Process Architectures*.

Meisels, I. et al. (1997). *Z/Eves 1.5 Reference Manual*. ORA Canada. TR-97-5493-03d.

Miller, T., Freitas, L., Malik, P., and Utting, M. (2005). CZT Support for Z Extensions. In *5th IFM*, number 3771 in LNCS, pages 227–245. Springer-Verlag.

Morgan, C. (1994). *Programming from Specifications*. Prentice-Hall.

Nordberg III, M. E. (1997). Default and Extrinsic Visitor. In *Pattern Languages of Program Design 3*. Addison Wesley.

Orni, R. et al. (1996). Two Computer Systems Paradoxes: Serialize-to-Parallelize, and Queuing Concurrent-Writes. Technical report, University of Maryland.

Probe (2000). *ProBE User's Manual version 1.28*. Formal Systems (Europe) Ltd.

Qin, S. C. et al. (2003). A Semantic Foundation of TCOZ in Unifying Theory of Programming. In *12th FM, Pisa*, number 3582 in LNCS. Springer-Verlag.

Roscoe, A. W. (1997). *The Theory and Practice of Concurrency*. Prentice-Hall.

Saaltink, M. et al. (2005). The Core Z/Eves API (DRAFT). Technical Report TR-99-5540-xxa, ORA Canada.

Schneider, S. (2002). *The B-Method—an Introduction*. Palgrave.

Smith, G. (2000). *The Object-Z Specification Language*. Kluwer Academic.

Utting, M. (2005). *Jaza User Manual and Tutorial*. University of Waikato, New Zealand.

Woodcock, J. and Davies, J. (1996). *Using Z*. Prentice-Hall.

Woodcock, J. C. P. and Cavalcanti, A. L. C. (2001). A Concurrent Language for Refinement. In *IWFM'01: 5th Irish Workshop in Formal Methods*.

Woodcock, J. C. P., Cavalcanti, A. L. C., Gaudel, M.-C., and Freitas, L. J. S. (2007). Operational Semantics for *Circus*. *Formal Aspects of Computing*. To appear.

Xavier, M. A. et al. (2006). Type Checking *Circus* Specifications. In *SBMF 2006: Brazilian Symposium on Formal Methods*, pages 105 – 120.